ISSN 2079-3316 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ №4(27), 2015, с. 313–340

УДК 510.252, 004.432.42

С. Д. Мешвелиани

# Программирование основ вычислительной алгебры на языке с зависимыми типами

Аннотация. В статье описываются главные черты разработанной автором на основе доказательного программирования библиотеки вычислительной алгебры. Обсуждается опыт доказательного программирования некоторых классических категорий вычислительной алгебры («группа», «кольцо» и так далее) на основе подхода конструктивизма, применения языка с зависимыми типами, построения машинно-проверяемых доказательств (dependent types, proof carrying code). Выявляются проблемы, связанные с этим подходом, и отмечаются дополнительные возможности, даваемые применением аппарата зависимых типов. В качестве инструмента используется функциональный язык Agda. Статья является продолжением вводной статьи автора в данном журнале за 2014 год.

*Ключевые слова и фразы:* конструктивная математика, алгебра, зависимые типы, функциональное программирование, Agda.

## Введение

В первой части [1] исследования применяется подход конструктивной математики [2] и конструктивной теории типов [3] к программированию некоторых методов вычислительной общей алгебры. На этих примерах описываются главные черты чисто функционального языка Agda [4, 5], воплощающего подход интуиционистской теории типов.

> Как только применены зависимые типы, алгоритмы (программы) естественным образом соединяются с *доказательствами*. И появляется возможность *доказуемого программирования*, когда заданные свойства

Исследование выполнено в рамках НИР «Методы анализа и верификации моделей вычислительных систем и алгебраических объектов на основе средств функционального и логического программирования» (№ г/р 01201354590) и «Методы и средства разработки эффективного программного обеспечения, ориентированные на вычислительные системы, построенные на основе микропроцессоров с многоядерной архитектурой, и формальные основы высокоуровневых языков программирования для суперкомпьютеров с гибридной архитектурой» (№ г/р 01201455360) (госзадание ФАНО России).

алгоритма автоматически проверяются компилятором (точнее — проверяльщиком типов). Более определённо, зависимые типы дают возможность:

- выражать свойство $P$ алгоритма в виде типа $T$ (зависящего от значений), причём конструкторы для $T$ задаются программистом,
- выражать доказательство свойства $P$ в виде функции, строящей любое значение в $T$,
- соединять в исходной программе алгоритм и доказательства его важнейших свойств (выбранных программистом), причём так, что наличие доказательств не замедляет вычисления,
- полагаться на автоматическую проверку доказательств,
- строить и надёжно автоматически проверять многие доказательства теорем в математике (ибо утверждения выражаются в виде зависимых типов).

Здесь описываются главные черты первой очереди библиотеки вычислительной алгебры, построенной автором на основе аппарата зависимых типов и программирования на языке Agda. Предполагается развитие этой программной системы.

Обсуждаемая здесь библиотека программ вычислительной алгебры DoCon-A написана на языке Agda, содержит описания алгоритмов и машинно-проверяемые доказательства их важнейших свойств, то есть автоматическую верификацию. Этот подход соответствует изложению алгоритмических методов алгебры в стиле учебника, вместе с определениями и доказательствами. Но разница состоит в том, что алгоритмы представлены в виде программ, определения и доказательства составляют часть программы и воспринимаются компилятором, доказательства являются полными и формальными, и проверяются компилятором.

Ближайшей целью работы является проверка практической возможности программирования на языке Agda некоторой начальной части библиотеки вычислительной алгебры, включающей полные доказательства. Эта проверка даёт ответы на следующие вопросы.

- Каковы выразительные возможности языка Agda: насколько математически адекватна запись определений и доказательств.
- Какова трудоёмкость составления машинно-проверяемых доказательств.
- Каков объём части исходной программы, воплощающей доказательства, по сравнению с объёмом соответствующего учебника, содержащего строгие (в обычном смысле) доказательства.
- Каковы затраты памяти и времени на проверку типов и компиляцию.

В конце статьи приводятся ответы на эти вопросы.

Последующее изложение опирается на

- понятия о чисто функциональном программировании, например, на языке Haskell, о «ленивом» способе вычисления,
- сведения о главных чертах языка и системы (proof assistant) Agda [4],
- изложение вводной статьи [1],
- общие понятия об алгебре [6],
- сведения о вычислительной алгебре [7, 8].

*Сокращения:* библиотеку программ DoCon-A будем сокращённо называть «библиотека».

«Стандартная библиотека» означает стандартную библиотеку языка Agda.

*О символах и именах в языке* Agda: приводимые в этой статье отрывки кода содержат математические символы и применяют систему языка Agda для задания идентификаторов. Эта система обозначений объясняется в [1], раздел 1.1.

## 1. Башня общих алгебраических структур

В настоящее время библиотека содержит определения и алгоритмы для некоторой иерархии классических общих алгебраических структур: от разрешимого множества (`DSet`), магмы (`Magma`) и полугруппы (`Semigroup`) до евклидова кольца (`EuclideanRing`) и поля (`Field`). Эта иерархия выражается графом, представленным ниже на рис. 1.

Пользователь библиотеки может её расширять, например, добавляя свои структуры к данной иерархии.

Нисходящие рёбра в этом графе означают наследование структур. Например, полугруппа должна иметь структуру магмы и ещё иметь добавочную структуру, выражающую ассоциативность операции. Два ребра, входящие в вершину `Ringoid` (пред-кольцо) выражают то, что пред-кольцо есть коммутативная группа по сложению (`_+_`) и магма по умножению (`_*_`). В этом графе пропущены некоторые законы, входящие в определения стуктур и в программу. Например: закон дистрибутивности для кольца.

Такой начальный набор структур выбран из тех соображений, что известны богатейшие теория и накопленная библиотека методов вычислений, определённых для этих структур (в основном это касается коммутативной алгебры в нётеровых областях). Например: решение линейной системы над евклидовым кольцом, арифметика области

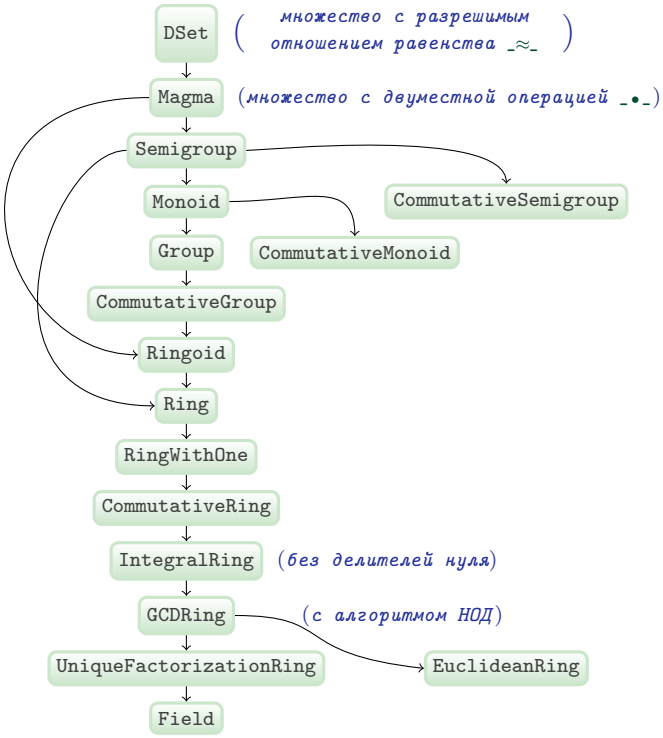


Рис. 1. Башня общих алгебраических структур

остатков по идеалу в кольце многочленов над евклидовым кольцом, разложение на простые множители многочлена над различными полями. Другой причиной выбора этой области является та, что ранее автором разработана на языке Haskell библиотека DoCon [9], которая посвящена в основном вычислениям в вышеназванных структурах. А библиотека DoCon-A разрабатывается как более адекватное выражение библиотеки DoCon. Усиление состоит в следующих двух главных дополнительных возможностях, даваемых аппаратом зависимых типов.

(1) Адекватное выражение конструкции алгебраической области, зависящей от динамически вычисляемых значений ([1], раздел 0.2).
(2) Возможность доказательного программирования с машинно проверяемыми доказательствами.

Причём даже при отсутствии части (2) получается существенное улучшение способа программирования.

## 2. Разрешимый сетоид

Сетоид определён в стандартной библиотеке языка Agda как множество (`Carrier`) с отношением `_≈_` равенства (эквивалентности) на нём. Это определение выражено в виде *типа* — записи (`record`). Отношение `_≈_` программируется пользователем для каждого случая сетоида. Например, программа для равенства многочленов может включать приведение подобных членов, так что, например, будет выполнено равенство `(x + 2 * x) ≈ (3 * x)`.

Там же в стандартной библиотеке определён разрешимый сетоид `DecSetoid` — сетоид с добавленным алгоритмом `_≟_` разрешения для отношения `_≈_`.

Библиотека ставит *разрешимый сетоид* в основу башни алгебраических структур, для которых задаются методы вычислений. Например, полугруппа определяется только на разрешимом сетоиде. Причина для такого подхода есть та, что при отсутствии разрешимого равенства (`_≟_`) для арифметики невозможны почти никакие алгоритмические методы. Например, при сложении двух квадратных многочленов окажется невозможным определить степень суммы.

## 3. Конечное перечисление и бесконечное множество

В алгебре особое место занимают конечные структуры; для них доказано много замечательных утверждений и разработаны многие особо эффективные алгоритмы. Свойство конечности алгебраической области в системе DoCon-A выражено в виде наличия перечисляющего списка для носителя (`Carrier`) области:

*Agda*

```
record FiniteEnumeration (A : Setoid) : Set
 where
 constructor finEnum'
 open Setoid A using (Carrier; _≈_)
 open Membership2 A using (AnyRepeats)

 field
   list   : List Carrier
   card   : ℕ
   proofs : (¬ AnyRepeats list) × IsFullList A list × length list ≡ card
```

(*здесь и ниже пропускаются некоторые мало-существенные подробности кода*).

В этой записи

`Set` есть множество всех типов (объяснения имеются в [4]) (так что допускаются функции, имеющие типы в качестве значений),

`A` есть параметр (аргумент) — сетоид, `Carrier` есть его носитель. Для каждого определённого пользователем примера структуры (в данном случае, записи `FiniteEnumeration`) пользователь должен задать программы вычисления значений полей этой структуры, то есть — частей, стоящих под ключевым словом `field` (в данном случае, значений `list`, `card`, `proofs`).

`list` есть перечисляющий список для носителя,

`card` есть мощность множества `Carrier` (относительно отождествления `_≈_`) — натуральное число.

`proofs` состоит из *свидетельств* (доказательств) для трёх утверждений (конъюнкция утверждений в конструктивной логике выражается в виде прямого произведения `_×_` типов):

(1) свидетельство для утверждения об отсутствии повторов в списке `list`;

(2) свидетельство для утверждения о том, что список `list` является полным для носителя;

(3) свидетельство для равенства мощности носителя длине списка `list`.

Каждое из этих утверждений выражено в виде задания соответствующего типа данных.

Например, приведём определение для второго утверждения:

*Agda*

```
IsFullList : (A : Setoid) → List (Setoid.Carrier A) → Set
IsFullList A xs =  (x : Setoid.Carrier A) → x ∈ xs
                                          where
                                          open Membership A using (_∈_)
```

.

Этот тип данных представляет утверждение

> «*Всякий элемент* `x` *носителя равен в смысле* `_≈_` *некоторому элементу списка* `xs`».

Отношение `_∈_` принадлежности списку определено в параметрическом модуле `Membership` из стандартной библиотеки.

Определение для первого утверждения мы пропускаем ради упрощения изложения.

Итак, видим, что определение конечного перечисления области выражено в виде типа данных — записи. Поля этой записи зависят как от «обычного» значения (натурального числа), так и от некоторых значений — свидетельств. Причём между этими значениями заданы зависимости. Это построение называется *зависимая запись* (dependent record) — частный случай *зависимого типа* (dependent type). Эта конструкция языка обладает такой математической выразительностью, какой нет в языках предыдущего поколения (например, в языках Haskell, ML).

### 3.1. Доказательно бесконечное множество

Конструктивное понятие бесконечности множества библиотека определяет в виде типа `Infinite` некоторых отображений из типа списков над носителем `C` сетоида:

*Agda*

```
Infinite : Set
Infinite = (xs : List C) → ∃ \x → x ∉ xs
```

.

Всякий элемент типа `Infinite` есть (алгоритмическое) отображение, которое для каждого списка `xs` над `C` выдаёт пару `(x , x∉xs)`, где `x` принадлежит `C` `(x : C)`, а `x∉xs` есть свидетельство того, что `x` не принадлежит списку `xs` в смысле равенства `_≈_` `(x ∉ xs)`.

*Пример:* библиотека использует следующее доказательство бесконечности множества натуральных чисел.

Алгоритм `getOutOfList` принимает список `xs : List ℕ`, находит `m = maximum (0 :: xs)` и выдаёт пару `(m' , m'∉xs)`, где `m' = m + 1`, а `m'∉xs` есть свидетельство того, что `m'` не принадлежит списку `xs`. Исходный текст этой функции

*Agda*

```
getOutOfList : (xs : List ℕ) → ∃ \x → x ∉ xs
```

занимает 20 непустых строк, и его главная часть представляет конструктивное доказательство того, что каждый элемент списка `xs` меньше числа `(maximum (0 :: xs)) + 1`. Доказательство использует функцию `maximum`, которая в библиотеке запрограммирована в отдельном модуле для общего случая полного разрешимого упорядочения (`DecTotalOrder`) и которая кроме максимума `m` для непустого списка `xs` выдаёт свидетельства для утверждений

```agda
  m ∈ xs,   All (\x → x ≤ m) xs
```

### 3.2. Множество с разрешённым свойством конечности

Следующий безымянный параметрический модуль выражает понятие алгоритмически разрешённого свойства конечности множества:

```agda
module _ (A : Setoid)
  where
  private C = Setoid.Carrier A
  open Membership A using (_∉_)

  Infinite :  Set
  Infinite =  (xs : List C) → ∃ \x → x ∉ xs

  data DecFiniteEnumeration : Set
       where
       finEnum  : FiniteEnumeration A → DecFiniteEnumeration
       infinite : Infinite            → DecFiniteEnumeration
```

Здесь тип данных `DecFiniteEnumeration` описывает все возможные свидетельства для конечности или бесконечности множества `C` в смысле равенства `_≈_`. Значение вида (`finEnum en`) из этого типа означает, что `en : FiniteEnumeration A` есть конечное перечисление для носителя `C` (как оно определено выше). Значение `(infinite inf)` из этого типа означает, что `inf` есть свидетельство бесконечности `C` (как оно определено выше).

## 4. Разрешимое множество

Библиотека ставит *разрешимое множество* (структура `DSet`) в основу башни алгебраических структур. Разрешимое множество определяется как запись из полей 1) разрешимый сетоид `decS`, 2) функция распечатки элемента в строку, 3) возможное разрешённое свойство конечности:

*Agda*

```
record DSet (decS : DecSetoid) : Set
  where
  open DecSetoid decS using (setoid; Carrier)

  field  showInstance : Show Carrier
         mbFiniteEnum : Maybe $ DecFiniteEnumeration setoid
```

Разрешимый сетоид `decS` даётся в аргументе записи. Программист задаёт функцию `showInstance` для распечатки всевозможных результатов. Для некоторых конструкторов типа она запрограммирована в библиотеке.

Тип поля `mbFiniteEnum` отличается от разрешённой конечности конструктором `Maybe` — «может быть». Значение вида (`just` (`finEnum en`)) этого поля означает, что `en` есть конечное перечисление для носителя.

Значение (`just` (`infinite inf`)) означает, что `inf` есть свидетельство бесконечности носителя.

Значение `nothing` означает: «не известно ни конечного перечисления носителя, ни конструктивного доказательства его бесконечности».

### 4.1. Формат «может-быть»

Введение формата «может-быть» для некоторых операций вызвано двумя причинами.

(1) Разрешимые множества появляются как конструкции над типами, зависящими от значений, а значения часто появляются в ходе вычисления, они могут быть неизвестны до начала выполнения программы. Наличие или отсутствие конечного перечисления зависит от этих значений. Проблема определения наличия конечного перечисления в зависимости от этих значений в общем случае не имеет алгоритмического решения. Например: определение конечности кольца остатков алгебры многочленов от переменных `[x, y, z]` по модулю данного набора уравнений (в зависимости от этих уравнений) может быть тяжёлой проблемой.

(2) Во многих структурах (скажем, в кольце) некоторые операции в том или ином случае этой структуры легко задать алгоритмом (скажем, `0`, `+`, `*`), а найти алгоритм для других операций этой структуры (скажем, поиск делителя нуля) является проблемой. Надо ещё учитывать, что случай структуры может зависеть от значений, которые могут быть неизвестны до начала выполнения программы.

Эти два затруднения разрешаются введением формата «может-быть» для некоторых операций.

## 5. Магма

Для сетоида `S`, двуместной операции `_•_` на его носителе `C` и элементов `a` и `b` из `C` *правое частное* `a` по `b` определяется в виде типа

*Agda*
```
RightQuotient : C → C → Set
RightQuotient a b =  ∃ (\q → (b • q) ≈ a)
```

В стандартной библиотеке знак существования `∃` имеет конструктивный смысл. Так, в данном случае частное есть пара `(q , e)`, где `q : C`, а `e` есть свидетельство для равенства `b • q ≈ a`.

Магма есть разрешимое множество с двуместной операцией (обозначаемой `_•_`) конгруентной относительно отношения равенства:

*Agda*
```
record Magma (upDS : UpDSet) : Set
 where
 private dS = UpDSet.dSet upDS
 open DSet dS using (≈equiv; _≈_) renaming (Carrier to C; setoid to S)
 ≈refl = IsEquivalence.refl ≈equiv    -- закон рефлексии равенства

 field  _•_            : Op₂ C
        •cong          : _•_ Preserves₂ _≈_⟶_≈_⟶_≈_
        mbCommutative  : Maybe $ DecCommutative S _•_
        divRightMb     : (x y : C) → Maybe $ Dec $ RightQuotient S _•_ x y

 •cong₁ : y : C → (\x → x • y) Preserves _≈_⟶_≈_
 •cong₁ x=x' =  •cong x=x' ≈refl

 _|_ : Rel C _
 x | y =  RightQuotient S _•_ y x     -- <<x делит y>>

 ...
```

Здесь четыре поля записи (под ключевым словом `field`) составляют определение магмы. Они суть отвлечённые операции. От программиста требуется задать реализацию этих операций в виде программ (функций) для каждого используемого случая этой структуры.

Так, в данном случае `_•_` есть операция магмы вообще, а для магмы натуральных чисел по умножению подставляется `_•_ = _*_`, и программируется умножение `_*_` для натуральных чисел.

Но имеются ещё дополнительные значения: типы данных и функции. Такие дополнительные сущности бывает удобно включать в общую структуру (запись), так как запись может быть использована как программный модуль. Эти дополнительные сущности получают реализацию уже в теле общей структуры и не требуют программирования для случаев этой структуры.

Продолжим объяснение записи `Magma`.

Поле `•cong` выражает закон конгруентности.

Поле `mbCommutative` выражает возможное свойство коммутативности операции `_•_`.

Поле `divRightMb` выражает (расширенную частичную) операцию деления в магме. Её итог вида `(just (yes quot))` означает, что `quot` есть правое частное, включающее свидетельство.

Итог вида `(just (no noQuot))` означает, что `noQuot` есть свидетельство отсутствия правого частного.

Итог вида `nothing` означает отсутствие сведений о частном `x` по `y` (в некоторых областях поиск частного является тяжёлой алгоритмической проблемой).

*Пример:* для магмы натуральных чисел по умножению имеем

*Agda*
```
divRightMb 6 2 =  just (yes (3 , 2*3≡6))
divRightMb 6 4 =  just (no noQuot-6-4)
```
,

где `2*3≡6 : 2 * 3 ≡ 6` и

*Agda*
```
Quot-6-4 :  ¬ RightQuotient natSetoid _*_ 6 4
```

суть соответствующие свидетельства.

Дополнительная функция `•cong₁` есть *лемма*, которая сопровождает определение магмы. Её реализация (в виде одной строчки) есть доказательство конгруентности операции `_•_` по первому аргументу. Выносить ли ту или иную лемму в отдельный модуль или оставить её в теле определяемой структуры — есть дело вкуса разработчика библиотеки.

Дополнительный тип `_|_` определяет отношение делимости на магме.

*О выборе набора операций:* поля `_•_` и `•cong` должны присутствовать в любой версии определения магмы, ибо они дают классическое определение. Набор других полей есть дело вкуса разработчика библиотеки алгебры. Например, можно было бы включить частичный атрибут порождающего множества элементов, функцию периода элемента, и так далее.

## 6. Отступление: up-структуры

Очередная техническая проблема такова:

выразить строение двух алгебраических структур, наследующих один и тот же случай некоторой структуры, так, чтобы избежать неприемлемо больших выражений для алгебраических структур.

Например: два векторных пространства над одним и тем же полем коэффициентов.

В математике такой проблемы нет: пишется просто «Пусть $U$ и $V$ суть векторные пространства над полем $F$».

В разрабатываемой библиотеке алгебраическая структура представляется записью (record), наследование структур представлено включением одной записи в качестве поля в другую запись. Такое представление выглядит простым и естественным, и другого разумного представления не видно. Но тогда необходимо выразить то отношение, что две записи имеют равные значения некоторого поля записи, где это значение есть некоторая алгебраическая структура (например, полугруппа). Задать алгоритмическое равенство на таких структурах не представляется возможным. Но нужное отношение просто выражается введением в объявление структуры аргумента в виде переменной. Например, выше запись `Magma` зависит от аргумента, обозначенного переменной `upDSet`. Тогда строки

*Agda*

```
M1 : Magma upDS
M2 : Magma upDS
```

объявляют что магмы `M1` и `M2` определены на одном множестве.

Далее: полугруппа получит в аргументе магму, группа получит в аргументе полугруппу, кольцо получит в аргументах коммутативную группу по сложению и полугруппу по умножению (имеющие общее множество), и так далее. Этот естественный подход ведёт к следующему неожиданному явлению. Например, в объявлении вида

*Agda*
```
E : EuclideanRing <аргументы>
```

каждая структура – аргумент должна сопровождаться её собственными аргументами. Уровень вложенности структур велик, и это приведёт к неприемлемо большому выражению для того, что в учебниках коротко выражается предложением «Пусть E есть евклидово кольцо».

И вот: применение up-структур позволяет убрать упоминание аргументов структур тогда, когда это нужно, и таким образом позволяет

- выразить наследование через включение структур,
- выразить общность наследуемой структуры для нескольких структур через подстановку в аргумент,
- сделать размер объявления структуры таким же малым, как в учебниках алгебры.

Техника up-структур состоит в следующем.

В аргументе записи `Magma` стоит структура `upDS : UpDSet`, которая до сих пор не описана и которая есть up-разновидность разрешимого множества (`DSet`):

*Agda*
```
record UpDSet : Set  where
                     field  decSetoid : DecSetoid
                            dSet      : DSet decSetoid

                     open DSet dSet public hiding (decSetoid)
```

Запись `DSet` зависит от аргумента типа `DecSetoid`, а в записи `UpDSet` значение этого аргумента подставляется из поля `decSetoid`. Соответственно, запись `UpDSet` не требует аргумента.

Далее:

`Magma` имеет аргумент `upDSet : UpDSet`,

`UpMagma` имеет поля `upDSet : UpDSet` и `magma : Magma upDSet`,

`Semigroup` имеет аргумент `upMagma : UpMagma`,

`UpSemigroup` имеет поля `upMagma : UpMagma` и `semigroup : Semigroup magma`.

и так далее: `Group` — `UpGroup`, `Ring` — `UpRing`, ...

up-структуры не имеют аргумента.

*Пример:* векторное пространство (структура `VectorSpace`) определено для *поля* `F : Field` коэффициентов и коммутативной группы `G` векторов.

(*Об обозначениях:* поле как алгебраическая структура выражается словом, набранным наклонным шрифтом — *поле*. А поле как существенная часть записи (record) выражается словом, набранным обычным шрифтом. )

В библиотеку понадобится включить определение общего понятия линейного отображения $f : U \to V$ пространств. Для этого необходимо выразить требование того, что они суть пространства над одним и тем же *полем* $F$. А естественное выражение нужного понятия берём из учебников алгебры: структура `VectorSpace` имеет аргументами (параметрами) структуру *поля* коэффициентов и структуру коммутативной группы $G$ векторов.

Например, для функции $f$ двух векторных пространств над одним полем объявление (сигнатура) выглядит так:

*Agda*

```
f : (upF : upField) → (upGU upGV : UpCommutativeGroup) →
    (U : VectorSpace upF upGU) → (V : VectorSpace upF upGV) → ...
```

(переменная `upF` входит трижды). Программа для этой функции может весьма просто извлечь все необходимые части из up-структур `upF`, `upGU`, `upGV`.

Заметим, что стандартная библиотека применяет другой подход. Но в её подходе появляются парные структуры: `Semigroup` — `IsSemigroup`, `Group` — `IsGroup`, и так далее. Причем обе части пары имеют одинаково объёмные выражения.

## 7. Отступление: отношение к стандартной библиотеке

Стандартная библиотека языка Agda тщательно продумана, содержит базовые определения и леммы присущие самым общим конструкциям: отношениям, отображениям, спискам, и так далее. Она очень полезна в применении. Она также имеет набор общих алгебраических структур: `Semigroup`, `Monoid`, `Group`, `AbelianGroup`, `Ring`, `CommutativeRing`.

По этой небольшой части две библиотеки пересекаются. DoCon-A импортирует эти стандартные структуры, переименовывая их добавлением окончания `-stdlib`. Например, `Group-stdlib`. Кроме того, DoCon-A предоставляет функции перевода в `stdlib` - структуры. Например, функция

*Agda*
```
ringWithOne-toStdlib : UpRingWithOne → Ring-stdlib
```

преобразует в кольцо с единицей стандартной библиотеки.

Различие на этом пересечении вызвано тем, что библиотека DoCon-A разрабатывается как продвинутое *приложение*, она содержит гораздо больше алгебраических структур и алгебраических методов (и будет расти). А общие со стандартом алгебраические структуры наделяются некоторыми дополнительными операциями. Это вызвано тем, что практическая алгоритмическая алгебра в отличие от классической учитывает требования удобства задания вычисления и сбережения объёма вычисления.

*Пример:* в классической теории *моноид* имеет две операции: `_•_` и $\varepsilon$ (единица). А библиотека DoCon-A добавляет в моноид операцию `_^_` обобщённого возведения в степень, вместе с двумя законами для неё:

*Agda*
```
∀ x n  ( x ^ 0 ≈ ε,    x ^ (1 + n) ≈ x • (x ^ n) )
```
.

Можно было бы вынести операцию `_^_` в отдельный модуль, использующий структуру классического моноида, и запрограммировать её через двоичный способ возведения в степень. Но решение в библиотеке более гибкое. Например, для аддитивного моноида натуральных чисел операцию `_^_` естественно реализовать как умножение `_*_`, и вычисление будет значительно быстрее, чем двоичный способ с повторным сложением.

В целом:

(1) отличия от классических определений вызваны необходимостью поддержки задания эффективных алгоритмов,
(2) всякая общая алгебраическая структура библиотеки влючает в себя все операции из классического определения.

## 8. Полугруппа, моноид

Продолжим описание представления алгебраических структур.

Полугруппа определяется как ассоциативная магма:

*Agda*

```
record Semigroup (upMg : UpMagma) : Set
 where
 private open UpMagma upMg using
                            (isEquivalence; setoid; _≈_; _•_; _|_; •cong₁)
         module FP≈ = FuncProp _≈_
         open EqR setoid renaming (_≈⟨_⟩_ to _≈[_]_)

 open IsEquivalence isEquivalence using () renaming (sym to ≈sym)

 field  •assoc : FP≈.Associative _•_     -- главная часть определения

 open UpMagma upMg public

 -- Лемма, добавленная <<на месте>> -------------------------------------

 |• : ∀ x y z → x | y → x | (y • z)
 |• x y z (q , x•q=y) =  (q • z , x•qz=yz)
                where
                x•qz=yz :  x • (q • z) ≈ y • z
                x•qz=yz =  begin  x • (q • z)   ≈[ ≈sym $ •assoc x q z ]
                                  (x • q) • z   ≈[ •cong₁ x•q=y ]
                                  y • z
                           □
```

Здесь используется определение ассоциативности `FP≈.Associative`, данное в стандартной библиотеке.

Лемма же утверждает и доказывает: “для любых `x`, `y`, `z`, если в полугруппе `x` делит `y`, то `x` делит `(y • z)`”.

Построение похожего доказательства объясняется в [1] (раздел 1.5).

Далее, единица в магме определяется функциями

*Agda*

```
 isUnity : C → Set
 isUnity e =  (x : C) → ((e • x) ≈ x) × ((x • e) ≈ x)

 Unity :  Set
 Unity =  ∃ \(e : C) → isUnity e
```

*Моноид* есть полугруппа, обладающая единичным элементом:

*Agda*

```
record Monoid (upSmg : UpSemigroup) : Set
 where
 open UpSemigroup upSmg using (_≈_; _•_)
                 renaming (Carrier to C; setoid to S; upMagma to upMg)

 field  unity : Unity S _•_

 ε : C
 ε = proj₁ unity

 field  invMb : (a : C) → Maybe $ Dec $ Inverse upMg ε a
        _^_   : C → ℕ → C
        ^-0   : ∀ x   → (x ^ 0) ≈ ε
        ^-suc : ∀ x e → (x ^ (suc e)) ≈ (x • (x ^ e))

 -- Леммы <<на месте>> ------------------------------------------------

 εlaw : (x : C) → (ε • x) ≈ x × (x • ε) ≈ x
 εlaw = proj₂ unity

 ε• : (x : C) → (ε • x) ≈ x
 ε• = proj₁ ∘ εlaw

 •ε : (x : C) → (x • ε) ≈ x
 •ε = proj₂ ∘ εlaw
```

.

Аргумент `upSmg` и операция `unity` (включающая свидетельство для закона единицы) составляют классическое определение моноида.

Определение `Inverse` обратного элемента подобно описанному выше определению `RightQuotient`.

`invMb` есть операция частичного обращения: некоторые элементы имеют обратные относительно умножения.

Возведение в степень `_^_` и его законы обсуждались выше.

Единица `ε : C` и закон умножения `εlaw` на неё извлекаются из данного `unity`.

Доказательства лемм `ε•` и `•ε` используют две проекции закона `εlaw`.

## 9. Группа, кольцо

Группа есть моноид с законом обращения:

*Agda*

```
record Group (upMon : UpMonoid) :  Set
 where
 monoid = UpMonoid.monoid upMon
 private open Monoid monoid using (setoid; _≈_; ≈equiv; _•_; •assoc;
                                                    unity; ε; •cong₁)
                              renaming (upMagma to upMg; Carrier to C)
         open EqR setoid renaming (_≈⟨_⟩_ to _≈[_]_)

 ≈sym      = IsEquivalence.sym ≈equiv
 εProofMap = proj₂ unity

 field  inverse : (a : C) → Inverse upMg ε a  -- главное в определении

 ------------------------------------ Дополнительные функции и леммы --
 _⁻¹ : C → C
 _⁻¹ = proj₁ ∘ inverse

 divide : (a b : C) → RightQuotient setoid _•_ a b
 divide a b  with  inverse b
 ...    | (i , (lProof , _)) = (i • a , qProof)
               where
               qProof : (b • (i • a)) ≈ a
               qProof = begin  b • (i • a)   ≈[ ≈sym $ •assoc _ _ _ ]
                               (b • i) • a   ≈[ •cong₁ lProof ]
                               ε • a         ≈[ proj₁ $ εProofMap a ]
                               a
                         □
 _/_ : Op₂ C
 x / y =  proj₁ $ divide x y

 open Monoid monoid public
```

Здесь классическая (постфиксная) операция обращения `_⁻¹` извлечена из операции расширенного обращения `inverse`.

Расширенное деление `divide` запрограммировано через операцию inverse, и оно включает доказательство свойства частного.

Классическое частное `x / y` извлекается из итога расширенного деления.

*Коммутативная группа* есть группа с добавленным законом коммутативности:

*Agda*

```
record CommutativeGroup (upG : UpGroup) :  Set
 where
 group = UpGroup.group upG
 private open Group group using (upSemigroup; upMonoid; _≈_; _•_)
         module FP≈ = FP _≈_

 field  commutative : FP≈.Commutative _•_

 commutativeMonoid : CommutativeMonoid upMonoid
 commutativeMonoid = commMonoid' commutative

 commutativeSemigroup : CommutativeSemigroup upSemigroup
 commutativeSemigroup = CommutativeMonoid.commutativeSemigroup
                                                    commutativeMonoid
 open Group group public
```

*Пред-кольцо* (Ringoid) есть коммутативная группа по сложению и магма по умножению, причём эти две структуры определены на одном разрешимом множестве:

*Agda*

```
record Ringoid (upA : UpCommutativeGroup) :  Set
  where
  +commGroup = UpCommutativeGroup.commutativeGroup upA
  open CommutativeGroup +commGroup using (_≈_; upDSet; ε)
                        renaming (Carrier to C; magma to +magma;
                                  monoid to +monoid; upGroup to  +upGroup)

  field  *magma : Magma upDSet
  -----

  0   = Monoid.ε +monoid     -- нуль пред-кольца
  _+_ = Magma._•_ +magma     -- сложение пред-кольца
  _*_ = Magma._•_ *magma     -- умножение пред-кольца

  +cong = Magma.•cong +magma   -- переобозначение закона конгруентности

  -_ : C → C                             -- взятие противоположного элемента
  -_ = CommutativeGroup._⁻¹ +commGroup

  ...
```

Структура пред-кольца введена для того, чтобы, например, охватить алгебры Ли: алгебра Ли является пред-кольцом относительно операций `_+_` и `[,]`, но не является кольцом.

Аргумент `upA` и операция `*magma` составляют ядро определения, а всё остальное извлекается из этих данных.

То условие, что группа `upA` и мультипликативная магма `*magma` определены над одним множеством, обеспечено

- импортом значения коммутативной группы `+commGroup` из `upA`,
- импортом значения `upDSet` через объявление `open` из `+commGroup`,
- объявлением `*magma : Magma upDSet`.

*Кольцо* есть пред-кольцо, в котором выполнен закон дистрибутивности и в котором мультипликативная магма является полугруппой:

*Agda*

```
record Ring (upRd : UpRingoid) :  Set
 where
 ringoid = UpRingoid.ringoid upRd

 open Ringoid ringoid using (setoid; _≈_; _≉_; *upMagma; 0; _+_; -_;
                                                          _-_; _*_)
                      renaming (Carrier to C)

 field  *semigroup : Semigroup *upMagma
 -----

 *upSemigroup = upSemigroup' *upMagma *semigroup

 open Power *upSemigroup using (^+)

 field  distributive  : _DistributesOver_ _≈_ _*_ _+_
        mbZeroDivisor : Maybe $ Dec $ HasZeroDivisor upRd

 IsNilpotent : C → Set
 IsNilpotent a =   a ≉ 0 × (∃ \(n : ℕ) → ^suc a n ≈ 0)
                           where
                           ^suc = × n → ^+ x (suc n) suc>0
 HasNilpotent :  Set
 HasNilpotent =  ∃ \a → IsNilpotent a
```

Понятия «делитель нуля» ( `ZeroDivisor`) и «пред-кольцо имеет делитель нуля» ( `HasZeroDivisor`) определены раньше в условиях пред-кольца. Частичная операция `mbZeroDivisor` программируется в каждом случае кольца так, чтобы по возможности разрешить: есть ли в данном случае делитель нуля. Если он найден, то он ставится в итог вместе с соответствующим свидетельством.

`IsNipotent` и `HasNipotent` суть общие определения того, что такое нильпотент (в кольце), и свойства «кольцо имеет нильпотент». Это определение опирается на функцию `^+` возведение в положительную степень в полугруппе.

## 10. Что ещё есть в библиотеке

Далее, таким же порядком библиотека определяет: кольцо с единицей (`RingWithOne`), коммутативное кольцо (`CommutativeRing`), кольцо без делителей нуля (`IntegralRing`), евклидово кольцо (`EuclideanRing`), *поле*, коммутативный моноид с законом сокращения, моноид с однозначным разложением на множители (`UniqueFactorizationMonoid`), кольцо с однозначным разложением на множители (`UniqueFactorizationRing`).

Доказываются формально (на языке Agda) многочисленные леммы об этих структурах — все они известны из учебников и просты с точки зрения математики. Суть этой части разработки состоит в нахождении общего удобного подхода к представлению алгебраических структур.

Пока наиболее сложным оказалось представление понятия кольца с однозначным разложением на множители. Оно опирается на конструкцию моноида ассоциированных классов для моноида ненулевых элементов и на действия над мультимножествами. Потребовалось далёкое расширение библиотеки обработки списков.

Уже реализованным методам и теоремам для евклидовых колец и колец с разложением на множители могут быть посвящены дальнейшие статьи.

## 11. Безотносительность

В ходе исследования выявилась любопытная практическая черта конструктивной математики: свойство *зависимости от свидетельства*, и соответственно, необходимость доказывать безотносительность (irrelevance).

Рассмотрим пример. *Поле* содержит операцию деления на ненулевой элемент:

*Agda*
```
C = Carrier
divideByNZ : (a b : C) → b ≉ 0# → RightQuotient setoid _*_ a b

div : C → (b : C) → b ≉ 0# → C
div a b b≉0 =  proj₁ $ divideByNZ a b b≉0
```
.

Пусть `p , p' : b ≉ 0#` суть два свидетельства неравенства `b` нулю. Им соответствуют значения частных: `q = div a b p`, `q' = div a b p'`.

Может ли программа пользоваться однозначностью частного — (`q ≈ q'`)?

С точки зрения классической математики: `b ≉ 0#` есть условие на область определения отображения. Оно может быть либо выполнено, либо не выполнено — булево значение. Если оно выполнено, то итог деления однозначно определён.

Но язык Agda поддерживает конструктивную математику: `b ≉ 0#` есть тип всех свидетельств неравенства `b` нулю, а их может оказаться бесконечно много. Во-первых, проверяльщик типов в выражении сигнатуры видит формальную зависимость от свидетельства. Во-вторых, в некоторых (других) примерах есть возможность запрограммировать функцию так, что она выдаст разные значения в зависимости от свидетельства.

То есть независимость частного от свидетельства в каждом примере либо не имеет места, либо требует доказательства. И программа, которая пользуется таковой независимостью, должна включать в себя доказательство теоремы независимости.

В данном примере доказательство независимости нетрудно извлечь из определения частного `RightQuotient`:

*Agda*

```
irrel-quot :  (a b : C) → (p p' : b ≉ 0#) → div a b p ≈ div a b p'
irrel-quot a b p p' =
                    let (q ,  bq≈a)  = divideByNZ a b p
                        (q' , bq'≈a) = divideByNZ a b p'

                        bq≈bq' :  b * q ≈ b * q'
                        bq≈bq' =  ≈trans bq≈a (≈sym bq'≈a)

                        q≈q' : q ≈ q'
                        q≈q' = cancelNonzeroLFactor b q q' p bq≈bq'
                    in
                    q≈q'
```

То есть из определения частного `RightQuotient` следует равенство `b * q ≈ b * q'`. Далее, *поле* наследует из структуры `IntegralRing` закон отсутствия делителя нуля; из него просто выводится закон `cancelNonzeroLFactor` сокращения на ненулевой множитель (вывод пропущен). Наконец, равенство `q ≈ q'` выводится применением последнего закона.

В других случаях подобная теорема независимости не выполнена (это зависит от задания определений в программе). И тогда, если математический смысл требует этой независимости, то её надо добавить как аксиому в одном из полей записи.

*Пример:* норма $|\_|$ в евклидовом кольце должна быть определена на ненулевых элементах. Поэтому библиотека даёт норме второй аргумент: свидетельство неравенства нулю первого аргумента. И оказывается необходимым добавить аксиому независимости нормы:

*Agda*

```
irrel-norm : x : C → nz nz' : x ≉ 0# → | x ,, nz | ≡ | x ,, nz' |
```

## 12. Перечень главных технических решений и особенностей

Перечислим описанные выше главные особые приёмы построения библиотеки.

В основу иерархии общих алгебраических структур поставлено множество с алгоритмом разрешения равенства `_≈_`.

Каждая общая алгебраическая структура представлена (зависимой) записью (dependent record). Наследование алгебраических структур выражено включением структуры как поля записи или как аргумента записи.

Применение up-структур позволяет убрать упоминание аргументов структур тогда, когда это нужно, позволяет выразить общность наследуемой структуры для нескольких структур и при этом сделать размер объявления структуры таким же малым, как в учебниках алгебры.

Классические алгебраические структуры наделяются некоторыми дополнительными операциями. Это вызвано тем, что практическая алгоритмическая алгебра имеет дополнительные требования по сравнению с классической алгеброй.

Для некоторых операций задан формат `Maybe` («может быть»). Этот приём применён к тем общим структурам, в некоторых случаях которых для одних операций даны алгоритмы, а для других алгоритмическое решение является проблемой.

Конгруентность алгебраических действий относительно равенства `_≈_` даётся или выводится в виде явного закона (тогда как в учебниках по математике она подразумевается).

Многие операции имеют *свидетельства* в качестве некоторых аргументов.

Безотносительность: в некоторых случаях независимость итога функции от выбора свидетельства даётся как закон, в остальных случаях она доказывается (если это существенно).

Свидетельства (доказательства) часто входят в итог функции (операции), и это вносит воспринимаемый компилятором смысл функции. Например, деление выдаёт частное `q` и свидетельство для решения уравнения (`b * q ≈ a`).

В объявления классических алгебраических структур добавлены некоторые леммы и функции «на месте» (используется свойство записи быть также программным модулем).

## 13. Выводы

### 13.1. Выразительность

На языке Agda алгебраические структуры, алгоритмы и конструктивные доказательства выражаются совершенно адекватно.

## 13.2. Объём исходной программы

Часть исходной программы, реализующая определения и доказательства, имеет объём текста, не сильно превышающий объем учебника, содержащего соответствующие строгие (в обычном смысле учебного материала) конструктивные доказательства.

## 13.3. Трудоёмкость построения доказательств

Приведём выдержку из статьи [1]:

"По текущему опыту проекта оказалось, что только трёх приёмов достаточно для построения удовлетворительно выглядящего доказательства:

(1) сведение (normalization),
(2) композиция функций,
(3) рекурсия.

Здесь (1) есть доказательство прямым вычислением (упрощением) к одному выражению, (2) соответствует введению леммы, (3) соответствует доказательству индукцией по построению данного. "

Пусть уже известно конструктивное доказательство того или иного утверждения строгое в обычном смысле учебного материала. Сколько времени может потребовать составление (полного формального) машинно-проверяемого доказательства для системы Agda? С одной стороны, построение машинно-проверяемых доказательств, — на опыте текущего состояния проекта, — очень трудоёмко. В 10 – 20 раз более трудоёмко, чем составление и отладка программы (без полной гарантии правильности) для самого метода вычисления. Это есть плата за верификацию.

С другой стороны, видно что после накопления некоторой достаточно широкой библиотеки лемм (особенно для списков) построение машинно-проверяемых доказательств в данной предметной области становится много легче.

Что касается возможных разработки и применения каких-то автоматических доказывателей (prover), то в текущем состоянии проекта не видно того участка, на котором их привлечение могло бы существенно облегчить построение доказательств.

## 13.4. Затраты компиляции

Текущая версия 2.4.2.4 системы Agda берёт для проверки типов в библиотеке DoCon-A примерно в 30 раз больше памяти и времени,

чем было бы естественно. Причина состоит в реализации интерпретатора вычислений на символьных выражениях типов, эта реализации требует улучшения. Этот недостаток интерпретатора скорее всего будет исправлен. Но пока непонятно, какие побочные явления это исправление может принести.

## 14. Реализация библиотеки

Выпуск 0.03 библиотеки DoCon-A доступен в WWW сети:
http://www.botik.ru/pub/local/Mechveliani/docon-A/

В этой статье описана часть опыта разработки следующего выпуска, который, вероятно, появится в 2016 году.

## Список литературы


[1] С. Д. Мешвелиани. «О зависимых типах и интуиционизме в программировании математики», *Программные системы: теория и приложения*, **5**:3(21) (2014), с. 27–50, URL: http://psta.psiras.ru/read/psta2014_3_27-50.pdf ↑313,315,316,328,337

[2] А. А. Марков, *Проблемы конструктивного направления в математике*, Сборник работ. Т. 2: *Конструктивный математический анализ*, Тр. МИАН СССР, т. **67**, Изд-во АН СССР, М.–Л., 1962, с. 8–14. ↑313

[3] P. Martin-Löef. *Intuitionistic Type Theory*, Bibliopolis, 1984. ↑313

[4] U. Norell, J. Chapman. *Dependently Typed Programming in Agda*, URL: http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf ↑313,315,318

[5] *Agda. A dependently typed functional programming language and its system*, Homepage of Agda, URL: http://wiki.portal.chalmers.se/agda/pmwiki.php ↑313

[6] А. И. Кострикин. *Введение в алгебру. Основы алгебры*, Наука. Физматлит, М., 1994. ↑315

[7] Дж. Коллинз, Р. Лоос. *Компьютерная алгебра. Символьные и алгебраические вычисления*, ред. Б. Бухбергер, Мир, М., 1986. ↑315

[8] Дж. Давенпорт, И. Сирэ, Э. Турнье. *Компьютерная алгебра*, Мир, М., 1991. ↑315

[9] S. D. Mechveliani. “Computer algebra with Haskell: applying functional-categorial-‘lazy’ programming”, International Workshop CAAP-2001 (Dubna, Russia, 2001), pp. 203–211, URL: http://compalg.jinr.ru/Confs/CAAP_2001/Final/proceedings/proceed.pdf ↑316


Рекомендовал к публикации *д.ф.-м.н. Н.Н. Непейвода*

*Об авторе:*

**Сергей Давидович Мешвелиани**

Старший научный сотрудник ИПС РАН. Занимается автоматизацией математических вычислений и рассуждений, функциональным программированием. Автор библиотеки вычислительной алгебры `DoCon`.

*e-mail:* mechvel@botik.ru

*Пример ссылки на эту публикацию:*

С. Д. Мешвелиани. «Программирование основ вычислительной алгебры на языке с зависимыми типами», *Программные системы: теория и приложения*, 2015, **6**:4(27), с. 313–340.

URL: `http://psta.psiras.ru/read/psta2015_4_313-340.pdf`

Sergei Meshveliani. *Programming basic computer algebra in a language with dependent types.*

Abstract. It is described the experience in provable programming of certain classical categories of computational algebra ("group", "ring", and so on) basing on the approach of intuitionism, a language with dependent types, forming of machine-checked proofs. There are detected the related problems, and are described certain additional possibilities given by the approach. The Agda functional language is used as an instrument. This paper is a continuation for the introductory paper published in this journal in 2014. (*In Russian*)

*Key Words and Phrases:* constructive mathematics, algebra, dependent types, functional programming, Agda.

*References*

[1] S. D. Meshveliani. "On dependent types and intuitionism in programming mathematics", *Programmnyye sistemy: teoriya i prilozheniya*, **5**:3(21) (2014), pp. 27–50 (in Russian), URL: http://psta.psiras.ru/read/psta2014_3_27-50.pdf

[2] A. A. Markov, "On constructive mathematics", *Problems of the constructive direction in mathematics*, Collection of articles. V. 2: *Constructive mathematical analysis*, Trudy Mat. Inst. Steklov., vol. **67**, Acad. Sci. USSR, M.–L., 1962, pp. 8–14 (in Russian).

[3] P. Martin-Löef. *Intuitionistic Type Theory*, Bibliopolis, 1984.

[4] U. Norell, J. Chapman. *Dependently Typed Programming in Agda*, URL: http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf

[5] *Agda. A dependently typed functional programming language and its system*, Homepage of Agda, URL: http://wiki.portal.chalmers.se/agda/pmwiki.php

[6] A. I. Kostrikin. *Introduction to Algebra. Fundamentals of algebra*, Nauka. Fizmatlit, M., 1994 (in Russian).

[7] R. Albrecht, B. Bukhberger, G. E. Collins, R. Loos (eds.), *Computer Algebra: Symbolic and Algebraic Computation*, Computing Supplementa, vol. **4**, 2nd ed., Springer, 1983, 283 p.

[8] J. H. Davenport, Y. Siret, E. Tournier. *Computer Algebra*, Seconde edition, Academic Press, 1993, 298 p.

[9] S. D. Mechveliani. "Computer algebra with Haskell: applying functional-categorial-'lazy' programming", International Workshop CAAP-2001 (Dubna, Russia, 2001), pp. 203–211, URL: http://compalg.jinr.ru/Confs/CAAP_2001/Final/proceedings/proceed.pdf

*Sample citation of this publication:*

Sergei Meshveliani. "Programming basic computer algebra in a language with dependent types", *Program systems: theory and applications*, 2015, **6**:4(27), pp. 313–340. (*In Russian*).

URL: http://psta.psiras.ru/read/psta2015_4_313-340.pdf